Planning with DLV

Theorem 1 (see *From Action Descriptions to Logic Programs*) shows that the edges of the transition system described by D can be determined from the list of the answer sets of logic program $lp(D)$. We will now define, for every positive integer n, a logic program $lp_n(D)$ whose answer sets correspond to paths of length n in the transition system described by D.

The language of $lp_n(D)$ has atoms of two kinds:

- the fluent names of D followed by (t) where $t = 0, \ldots, n$, and
- the action names of D followed by (t) where $t = 0, \ldots, n-1$.

Program $lp_n(D)$ consists of the following rules:

(i) for every static law

$$\textbf{caused } F \textbf{ if } L_1 \wedge \cdots \wedge L_m$$

in D, the rules

$$F(t) \leftarrow not\ \overline{L_1(t)}, \ldots, not\ \overline{L_m(t)}$$

for all $t = 0, \ldots, n$,

(ii) for every dynamic law

$$\textbf{caused } F \textbf{ if } L_1 \wedge \ldots \wedge L_m \textbf{ after } L_{m+1} \wedge \ldots \wedge L_n$$

in D, the rules

$$F(t+1) \leftarrow\ not\ \overline{L_1(t+1)}, \ldots, not\ \overline{L_m(t+1)}, L_{m+1}(t), \ldots, L_n(t)$$

for all $t = 0, \ldots, n-1$,

(iii) the rules

$$\neg B \leftarrow not\ B,$$
$$B \leftarrow not\ \neg B$$

for all atoms B that end with (0) or include an action name.

For instance, if D is

$$\begin{array}{l} \textbf{default } P \\ A \textbf{ causes } \neg P. \end{array} \tag{1}$$

then $lp_n(D)$ is

$$\begin{array}{ll} P(t) \leftarrow not\ \neg P(t) & (0 \leq t \leq n), \\ \neg P(t+1) \leftarrow A(t) & (0 \leq t < n), \\ \neg P(0) \leftarrow not\ P(0), & \\ P(0) \leftarrow not\ \neg P(0), & \\ \neg A(t) \leftarrow not\ A(t) & (0 \leq t < n), \\ A(t) \leftarrow not\ \neg A(t) & (0 \leq t < n). \end{array} \tag{2}$$

Program $lp_1(D)$ differs from $lp(D)$ in that every static law in D generates two rules of this program $(t = 0, 1)$ rather than just one. For instance, if D is (1) then $lp_1(D)$ contains one additional rule:

$$P(0) \leftarrow not\ \neg P(0).$$

These additional rules, generally, eliminate some of the answer sets for $lp(D)$ that do not correspond to the edges of the transition system described by D. Accordingly, Theorem 2 below provides a stronger characterization of the correspondence between edges and answer sets than Theorem 1.

For any set X of literals in the language of $lp_n(D)$, by X^f we denote the set of literals in X that include a fluent name; X^a stands for the set of literals in X that include an action name. We say that X is *complete* if, for every atom B, $B \in X$ or $\neg B \in X$.

Theorem 2. *A complete consistent set X of literals is an answer set for $lp_n(D)$ iff, for every $t = 0, \ldots, n-1$, the triple*

$$\langle\, \{L : L(t) \in X^f\}, \{L : L(t) \in X^a\}, \{L : L(t+1) \in X^f\}\,\rangle$$

is a transition causally explained by D.

The completeness requirement in the statement of Theorem 2 can be dropped if we extend $lp_n(D)$ by adding the constraints

$$\leftarrow not\ B, not\ \neg B$$

for all atoms B. In many cases, the program has no incomplete answer sets even without these constraints.

By instructing DLV to find the answer sets for program (2) we can find all paths of a given length in the transition system described by (1). This program can be represented as an input for DLV as follows:

```
p(T) :- not -p(T), #int(T).
-p(T1) :- a(T), #succ(T,T1).
-p(0) :- not p(0).
p(0) :- not -p(0).
-a(T) :- not a(T), #int(T), T != #maxint.
a(T) :- not -a(T), #int(T), T != #maxint.
```

If, instead of all paths, we want to find the paths that satisfy some additional conditions, we can accomplish this by adding appropriate constraints.

Assignment DLV-5. Use DLV to find all paths of length 3 in the transition system for (1) that lead from state P to state $\overline{P}$.

The last exercise can be viewed as the use of DLV to solve a planning problem: Given that the effects of actions $\overline{A}$ and A on fluent P are defined by action description (1) and that the initial state is P, find all plans of length 3 that make P false. To suppress displaying all information about the plan except for the list of actions to be performed, we can specify the option `-pfilter=a` on the command line. To cause DLV to terminate after finding one plan, we would specify the option `-n=1` .

Assignment DLV-6. The effects of elementary actions *Load* and *Shoot* on fluents *Loaded* and *Alive* are described in $\mathcal{C}$ as follows:

$$
\begin{array}{l}
\mathit{Load}\ \textbf{causes}\ \mathit{Loaded} \\
\textbf{nonexecutable}\ \mathit{Load}\ \textbf{if}\ \mathit{Loaded} \\
\mathit{Shoot}\ \textbf{causes}\ \neg \mathit{Alive} \\
\mathit{Shoot}\ \textbf{causes}\ \neg \mathit{Loaded} \\
\textbf{nonexecutable}\ \mathit{Shoot}\ \textbf{if}\ \neg \mathit{Loaded} \\
\textbf{inertial}\ \mathit{Loaded}, \neg \mathit{Loaded}, \mathit{Alive}, \neg \mathit{Alive}
\end{array}
$$

Initially *Loaded* is false and *Alive* is true. Use DLV to find the shortest plan for achieving the state in which *Loaded* is true and *Alive* is false (and to verify that there is indeed no shorter plan).

This planning method has two limitations.

First, if the given initial conditions do not provide a complete characterization of the initial state then we cannot claim that the plan found using this method succeeds *for every initial state* satisfying the initial conditions. In this sense, the method is not applicable to a planning problem if its initial conditions are incomplete.

Assignment DLV-7. Consider the modification of the planning problem from the previous assignment in which the initial value of *Alive* is not specified. Use DLV to find all solutions of length 3. Which of them are guaranteed to succeed for each of the possible initial states?

Second, if some of the actions have nondeterministic effects then we cannot claim that the plan found using this method succeeds *for all possible outcomes* of these actions. In this sense, the method is not applicable to a planning problem if its actions are nondeterministic.

Assignment DLV-8. Consider the modification of the planning problem from Assignment DLV-6 in which the causal law

$$\mathit{Shoot}\ \mathbf{causes}\ \neg \mathit{Alive}$$

is replaced with

$$\mathit{Shoot}\ \mathbf{may\ cause}\ \neg \mathit{Alive}.$$

Recall that, according to *Action Languages*, this is an abbreviation for

$$\mathbf{caused}\ \neg \mathit{Alive}\ \mathbf{if}\ \neg \mathit{Alive}\ \mathbf{after}\ \mathit{Shoot}.$$

(a) Use DLV to determine how this modification affects the transition system described by the given action description. (b) Determine how this modification affects the solution to the planning problem produced by DLV.

We should also keep in mind that this planning method may produce plans with concurrently executed elementary actions. Concurrency can be eliminated or limited by adding appropriate constraints.

Consider, for instance, the domain consisting of 5 lightswitches:

$$\begin{array}{l} \mathit{Toggle}(l)\ \mathbf{causes}\ \mathit{On}(l)\ \mathbf{if}\ \neg \mathit{On}(l) \\ \neg \mathit{Toggle}(l)\ \mathbf{causes}\ \neg \mathit{On}(l)\ \mathbf{if}\ \mathit{On}(l) \\ \mathbf{inertial}\ \mathit{On}(l), \neg \mathit{On}(l) \end{array}$$

($l \in \{L1, \ldots, L5\}$). Initially all lights are on; the goal is to turn all of them off. This planning problem can be given to DLV as the file

```
switch(l1). switch(l2). switch(l3). switch(l4). switch(l5).

on(L,T1) :- switch(L), toggle(L,T), -on(L,T), #succ(T,T1).
-on(L,T1) :- switch(L), toggle(L,T), on(L,T), #succ(T,T1).

on(L,T1) :- switch(L), not -on(L,T1), on(L,T), #succ(T,T1).
-on(L,T1) :- switch(L), not on(L,T1), -on(L,T), #succ(T,T1).
```

```
on(L,0) :- switch(L), not -on(L,0).
-on(L,0) :- switch(L), not on(L,0).

toggle(L,T) :- switch(L), #int(T), T!= #maxint, not -toggle(L,T).
-toggle(L,T) :- switch(L), #int(T), T!= #maxint, not toggle(L,T).

:- switch(L), -on(L,0).
:- switch(L), on(L,#maxint).
```

This formalization allows DLV to find a one-step solution in which all switches are toggled concurrently. To find a solution with actions executed one at a time, we can add the constraint

```
:- toggle(L1,T), toggle(L2,T), L1<L2.
```

Assignment DLV-9. For this planning problem, use DLV to find a shortest solution that does not involve toggling more than two switches at a time.